Encoding DCC in Haskell

Maximilian Algehed
algehed@chalmers.se
Chalmers University of Technology, Sweden

Alejandro Russo
russo@chalmers.se
Chalmers University of Technology, Sweden

ABSTRACT

The seminal work on the Dependency Core Calculus (DCC) [1] shows how monads not only can be used for embedding effects in purely functional languages but also to statically track data dependencies. Such types of analysis have applications in research areas like security, partial evaluation, and slicing, where DCC plays the role of a unifying formalism. For a Haskell programmer, putting DCC into practice raises many interesting conceptual and implementation concerns. Specifically, DCC uses a non-standard bind operator, i.e., with a different type signature than that provided by monads. In fact, embedding such non-standard bind operator opens the door for many design decisions. Furthermore, it is unclear if DCC extends to traditional methods used by Haskell programmers to handle effects (such as monad transformers). In this work, we describe a novel encoding of DCC in Haskell, with a focus on its use for security—although our results also apply to the other domains. We address the concerns mentioned above and show how our implementation of DCC can be seen as a direct translation from its typing rules via the use of *closed type families* and *type classes*—two advanced type system features of Haskell. We also analyze what kind of effects DCC is compatible with and which ones it cannot secure. We also derive an alternative formulation of DCC based on *fmap* and a corresponding non-standard *join*.

CCS CONCEPTS

• **Security and privacy → Information flow control**; **Formal security models**; • **Software and its engineering → Software libraries and repositories**;

KEYWORDS

Dependency Core Calculus, Information-Flow Control, Haskell

1 INTRODUCTION

Building applications which preserve privacy of data is an open and difficult problem. It is tempting to think that applying some sort of mechanism to grant or deny access to information will be sufficient to preserve secrets; *this is not the case*. Often, code is required to have access to sensitive data in order to deliver its functionality, but what we would like to know is if the code treats private data adequately, i.e., it does not leak it! The following example illustrates this point. It involves developers Alice and Bob, who are collaboratively writting a web application for *something very important* (we leave that up to the imagination of the reader).

Example 1.1. In an effort to make sure users' private data is kept safe, the web application is implemented using separated *public* and *private* databases. While implementing the user registration page, Alice has requested Bob's help in managing what data to commit to the database. Alice writes the following code.

Alice

```
webApp :: IO ()
webApp = do
  ...
  formData ← getUserRegisterForm
  let (pubDBEnts, privDBEnts) = mkDBEntries formData
  commitToPublicDB pubDBEnts
  commitToPrivateDB privDBEnts
  ...
```

The symbol :: is used to describe the type of terms in Haskell. The function *getUserRegisterForm* retrieves the registration page data from the user and the ... denotes some part of the code that is not relevant for the point being made. On Alice's request, Bob has written the function *mkDBEntries*[1]. We use the standard list syntax $[x_1, x_2, \cdots, x_n]$ for describing extensible lists and $[a]$ for denoting the type of lists of elements of type a.

Bob

```
mkDBEntries :: FormData → ([DBEntry], [DBEntry])
mkDBEntries form =
  ([screenName form, userEmail form, userName form],
   [hash (password form)])
```

What Bob does not know is that the name (*userName form*) of the user is not meant to be public information. This fact was hidden deep in a design document somewhere, but Bob was too lazy to read that document.

Information-Flow Control (IFC) is a research area dedicated to providing guarantees that private data is kept confidential by applying programming languages techniques [12, 20]. Haskell is one of a few a languages capable of guaranteeing IFC through libraries [15]. As long as applications adhere to the library's API, they are guaranteed to keep private data confined [6, 18, 19, 26]. These IFC libraries enforce security in the presence of many advanced programming languages features (e.g., concurrency) and often intertwine effect-free and effectful computations into a single monad.

[1] To simplify the presentation we take the type *DBEntry* to be *String* and consequently *hash* :: *String* → *String*. While a real database library will have a much richer representation of database entries, this simplification has no impact on the point being made.

PLAS 2017, October 30 2017, Dallas, Texas

ACM ISBN 978-1-4503-5099-0/17/10…$15.00
https://doi.org/https://doi.org/10.1145/3139337.3139338

$$\frac{\Gamma \vdash x : a}{\Gamma \vdash \texttt{return}_\ell\ x : T_\ell(a)}$$

$$\frac{\Gamma \vdash e : T_\ell(a) \quad \Gamma \vdash f : a \to s \quad \ell \preceq s}{\Gamma \vdash e \texttt{ >>>= } f : s}$$

Figure 1: Core typing rules for DCC.

$$\text{(R1)}\ \frac{\ell \sqsubseteq \ell'}{\ell \preceq T_{\ell'}(s)} \qquad \frac{\ell \preceq s \quad \ell \preceq t}{\ell \preceq (s \times t)} \qquad \frac{\ell \preceq t}{\ell \preceq (s \to t)}$$

$$\text{(R2)}\ \frac{\ell \preceq s}{\ell \preceq T_{\ell'}(s)} \qquad \text{(R3)}\ \frac{}{\ell \preceq ()}$$

Figure 2: The $\preceq$-relation

In this paper, we take a different approach by building an IFC library based on DCC [1]—a calculus designed to secure *pure computations*. We show how it is possible to decouple the core (pure) security concerns of applications from its (monadic) effects. As illustrated by the example above, getting IFC right is a tricky business and we are going to show how DCC can help us get it right!

2 BACKGROUND

Attacker model. We adopt the same attacker model as the one considered in the original DCC work [1]: *potentially malicious or careless programmers provide code that manipulates secret data; data which should not be leaked into public channels.* This choice is also aligned with the attacker model considered by other IFC libraries (e.g., LIO [26] and MAC [18]). The security policy that our implementation will enforce is *non-interference* [11]. Similarly to previous work [4, 30], we focus on the terminating fragment of DCC, thus ignoring leaks due to abnormal termination of programs—a security condition known as *termination-insensitive non-interference*. We also consider the treatment of other covert channels [14] like power consumption or memory footprint out of scope.

Dependency Core Calculus. DCC is a variant of the lambda calculus which features sums, products, and a family of identity monads, written T, which are indexed by elements of a lattice. For the purpose of this paper, elements of the lattice represent security levels denoting the sensitivity of data [8]. Importantly, the order relation of the lattice indicates how information is allowed to flow: $\ell \sqsubseteq \ell'$ dictates that information with sensitivity ℓ can flow into entities of sensitivity ℓ'. For simplicity, we restrict ourselves to the classic two-point lattice with levels H (for high or private) and L (for low or public), where $H \not\sqsubseteq L$ is the only disallowed flow.

The key property of DCC is that, once a value gets *protected* (wrapped) by T_ℓ, any computation that depends on it is forced to reside in the same or a more sensitive family member of T. That is to say that, once data is considered private, it may only influence private computations. In contrast, public data is allowed to influence both private and public computations. To capture this property, DCC provides a special typing rule for *bind*—after all, this is the operator which extracts values from computations of type T_ℓ. Such a typing rule prevents, for instance, public computations, of type $T_L(a)$, from depending on sensitive ones, of type $T_H(a)$.

Figure 1 shows the typing rules for `return` and bind, respectively. We denote bind as (>>>=) [2]—observe the extra > character compared to the bind operation (>>=) from Haskell. The notation $\Gamma \vdash e : \tau$ denotes the judgement "given the assumptions in Γ, the expression e has type τ", where Γ is on the form $a_0 : \tau_0,\ a_1 : \tau_1, \ldots$ for some variables $a_0, a_1, \ldots$ and types $\tau_0, \tau_1, \ldots$. The (>>>=) operator takes a computation m and a function f and computes the result of m and applies f to that result, yielding a new computation. For a functional programmer familiar with monads, DCC's bind has the *non-standard type*

$$\ell \preceq s \Rightarrow T_\ell(a) \to (a \to s) \to s,$$

where the type s is subject to the restriction $\ell \preceq s$ (explained below), rather than the conventional type

$$T_\ell(a) \to (a \to T_\ell(b)) \to T_\ell(b).$$

In Section 3, we will show that (>>>=) subsumes the standard bind operator, we therefore refer to it as *super-bind* from now on.

The typing rules for $return_\ell$ and (>>>=) differ from the traditional typing rules for monadic operations in two ways. Because DCC is a label-monomorphic calculus, the primitive $return_\ell$ features a sub-index ℓ denoting the family member which protects the given argument [3]. Other than that, this primitive is completely standard. Intuitively, the rule for super-bind forces the result to be protected at the same, or higher, level as the input. This last invariant is enforced by the side condition $\ell \preceq s$ (pronounce "s is protected at ℓ").

Figure 2 formally introduces the $\preceq$-relation. The judgement $\ell \preceq s$ is defined by structural induction on s:

- $T_{\ell'}(s)$ is protected at ℓ when $\ell \sqsubseteq \ell'$ or s is protected at ℓ.
- $(s \times t)$ is protected at ℓ when s and t are both protected at ℓ.
- $s \to t$ is protected at ℓ when t is protected at ℓ.
- $()$ is protected at ℓ.

This definition is very similar to the original definition by Abadi et al., it only differs in the rule `(R3)`, which was first introduced by Tse and Zdancewic in [31].

We provide some examples to clarify how super-bind works.

Example 2.1 (Respecting the security lattice). The super-bind allows computations to be built as long as they follow the order-relation described by the security lattice. For instance, super-bind type-checks if the computation stays within the same level [4].

$$x : T_L(Bool) \vdash x \texttt{ >>>= } \lambda y.\ return_L\ \neg y : T_L(Bool)$$

Likewise, super-bind type-checks if the result is more sensitive than the input.

$$x : T_L(Int) \vdash x \texttt{ >>>= } \lambda n.\ return_H\ n : T_H(Int)$$

[2] In the seminal work on DCC, the authors introduce bind as a let-like binding, **bind** $x = e$ **in** e'. Instead, we denote it as (>>>=) since it looks more familiar for a Haskell programmer. This choice is, however, only for improving the presentation.

[3] Observe, however, that sub-indexes are not strictly necessary since types can provide those indexes too.

[4] Technically, we take the type *Bool* to denote the DCC sum type () + (), and type *Int* to denote finite size integers represented as products of *Bools*.

```
{−# LANGUAGE DataKinds #−}
{−# LANGUAGE TypeFamilies #−}
data Lattice = L | H
type family ℓ ⊑ ℓ′ where
  H ⊑ L = False
  _ ⊑ _ = True
```

Figure 3: A lattice implemented with type families.

Observe that we started with a public value and end up with a sensitive one. However, downgrading the sensitivity of the data is disallowed. For example, the following program is ill-typed as $H \not\sqsubseteq L$.

$$return_H\ 42 >>>= \lambda n.\ return_L\ n$$

Example 2.2 (Statically predicted values). DCC leverages types to identify certain values which are independent of secrets. This aspect provides a more flexible super-bind, i.e., a super-bind which does not demand protection on the resulting value when it is unitary. For instance, code below type-checks in DCC.

$$\emptyset \vdash return_H\ 42 >>>= \lambda n.() : ()$$

The symbol $\emptyset$ denotes the empty typing environment. By examining the type of $\lambda n.()$, it is known that the produced value is unit and independent of the sensitive information—thus, it type-checks!

In a similar manner, protecting products is simply protecting the two components. After all, it is statically known that there are only two components in a product, so this fact is independent of secrets! The next piece of code is therefore well-typed.

$$\emptyset \vdash return_H\ 41 >>>= \lambda n.(return_H\ (n+1), ()) : (T_H\,\text{Int}, ())$$

Example 2.3 (Sums). Unlike products, super-bind cannot be flexible when it comes to sums. To protect a sum, it is not enough to just protect the content of their options. This is mainly due to the fact that the type of sums does not reflect the option being taken—thus, it can be misused to reveal information. The following ill-typed program shows that.

$$return_H\ s >>>= \lambda n.\ \textbf{if}\ n == 0\ \textbf{then}\ inl(return_H\ 1)\ \textbf{else}\ inr(return_H\ 0)$$

If this code were well-typed, it would be enough to observe if the returned value is a left (`inl`) or right (`inr`) injection to know if the secret is zero—despite the content of the options being protected.

3 EMBEDDING DCC IN HASKELL

In this section, we show how to implement DCC in Haskell[5] by leveraging on some of the recently added features of its type system. Importantly, the obtained code can be seen as a syntactic translation of the rules described in Figures 1 and 2, which gives us not only an elegant code but also confidence about its correctness.

[5]The code from this paper is available online at https://github.com/MaximilianAlgehed/DCC

```
  -- definition of T as an abstract data type
data T (ℓ :: Lattice) a = T^TCB {unT^TCB :: a}
  -- type signatures
return :: a → T ℓ a
(>>=) :: (ℓ ⪯ s)~True ⇒ T ℓ a → (a → s) → s
  -- implementation
return = T^TCB
t >>= f = f (unT^TCB t)
```

Figure 4: Core DCC primitives

```
type family ℓ ⪯ t where
  ℓ ⪯ (T ℓ′ s) = ℓ ⊑ ℓ′ ∨ ℓ ⪯ s
  ℓ ⪯ (s, t)   = ℓ ⪯ s ∧ ℓ ⪯ t
  ℓ ⪯ (s → t)  = ℓ ⪯ t
  ℓ ⪯ ()       = True
  ℓ ⪯ s        = False
```

Figure 5: The ⪯-relation as a closed type family.

Security lattice. Because we want to use Haskell's type system to implement the security features of DCC, we need to represent our lattice at the type level. Figure 3 shows our implementation for the two-point lattice. It uses Haskell's extensions for *closed type families* [10] and *data kinds* [35]. In a nutshell, closed type families are no more than type-level functions. The code defines such a function by introducing the type-level $\sqsubseteq$-relation on two arguments ℓ and ℓ'. The `DataKinds` extension, on the other hand, is not strictly necessary but allows booleans and security level constructors (L and H) to be promoted to the type level—hence, it yields more elegant and self-descriptive code. In other words, constructors L and H belonging to the data type *Lattice* can be interpreted as types too! In fact, the equations in Figure 3 involve types (and not terms) L, H, *True*, and *False*. The key part of the implementation is that the equations capture the expected security policy. Namely, as pointed out in Section 2, $H \not\sqsubseteq L$ is the only disallowed flow. It is also possible to encode more complex (potentially infinite) security lattices using type families [6]—e.g., the lattice induced by *disjunction category labels* [25], where security levels express concerns of mutually distrusting principals.

Dependency tracking. The key component necessary to obtain DCC-style dependency tracking is the implementation of the T-monad family and its corresponding operations. Figure 4 shows our Haskell definitions for T, return ($\underline{return}$), and super-bind ($\gg\!=$). At first glance, the code looks almost like a mere syntactic translation of the typing rules from Figure 1—except for the equality type constraint $(\ell \preceq s) \sim True$ (explained below). Just as in DCC, the type constructor T is a family of identity monads indexed by a security level ℓ coming from the kind *Lattice*. The definition of T involves "constructor" $T^{TCB} :: a \to T\ \ell\ a$ and the "destructor" $unT^{TCB} :: T\ \ell\ a \to$

a. These component are part of the *trusted computing base* or TCB (observe the supra-index). Consequently, it is not accessible to the users of our DCC implementation but is part of its internals. Otherwise, for instance, access to the constructor T^{TCB} would allow attacker's code to pattern match on a value $v :: T\ H\ a$ to simply extract the secret value (e.g., **case** v **of** T^{TCB} *secret* $\rightarrow$ *secret*).

Because Haskell is polymorphic, the type of $\underline{return}$ can capture the typing rules for the entire family of $return_\ell$, for any given ℓ. Similarly, the implementation of $(\ggg\!=)$ has almost the same type $(\ell \preceq s) \sim True \Rightarrow T\ \ell\ a \rightarrow (a \rightarrow s) \rightarrow s$ as the super-bind $(>>>=) : T\ l\ a \rightarrow (a \rightarrow s) \rightarrow s$ in DCC. The difference being that, as s is restricted by $\ell \preceq s$ in DCC, our implementation requires discharging the type equality constraint $(\ell \preceq s) \sim True$ to enforce the same characteristic over s. More specifically, the constraint $(\ell \preceq s) \sim True$ means that the application of the type-level function $\ell \preceq s$ evaluates to (the type) *True*. In the whole implementation, $(\ggg\!=)$ is the only operation which utilizes unT^{TCB}, and therefore is the *only primitive that we must trust to work as expected*—clearly, we must trust our type system and type declarations too! As the Haskell language permits certain unsafe operations[6] which might break type-safety and data encapsulation, we also rely on `SafeHaskell` [28]—an extension of the Haskell compiler GHC which restricts the use of unsafe features to trusted code. We now turn our attention to the type family implementing the $\preceq$-relation.

Protecting data. Figure 5 encodes the $\preceq$-relation of DCC as a closed type family with the same name. The code is almost an entirely syntactic translation of the $\preceq$-relation from Figure 2 except for the first two equations which use type-level versions of operators $\wedge$ and $\vee$. However, these cases follow directly from the structure of the $\preceq$-relation. For instance, the first equation collapses rules `(R1)` and `(R2)` into one disjunction. The rest of the cases are self-explanatory and therefore we omit describing them[7].

Why closed type families? Haskell supports type families, i.e., type-level functions, both in an open and closed variant. An open type family allows equations defining it being scattered throughout the code base. The type family definitions for the $\sqsubseteq$- and $\preceq$-relations are instead closed. This means that (i) equations may not be added outside the initial ones, and (ii) equations are always tried in order of appearance. Condition (i) is essential to guarantee security. Otherwise, for instance, if these type families were not closed, an attacker (or careless programmer) could inadvertently add the equation $H \sqsubseteq L = True$, or $\ell \sqsubseteq s = True$—thereby nullifying all security properties given by $(\ggg\!=)$. In contrast, condition (ii) is essential to avoid being overly restrictive. For instance, if equation $\ell \preceq s = False$ were to trigger first, many programs would be unnecessarily classified as insecure.

3.1 Revisited example

Let us see how the solution to Alice and Bob's problem can be translated from the theoretical calculus to Haskell.

Example 3.1. Alice starts by defining the security levels of her data as follows.

[6] like $unsafePerformIO :: IO\ a \rightarrow a$ and $unsafeCoerce :: a \rightarrow b$

[7] We note that, while the implementation requires the extension *UndecidableInstances*, the definition of the type family is always terminating.

```
instance Monad (T ℓ) where
  return :: a → T ℓ a
  return = return
  (>>=) :: T ℓ a → (a → T ℓ b) → T ℓ b
  (>>=) = (>>>=)
```

Figure 6: Monad instance for $T\ \ell$

Alice

```
data FormData = FD {password    :: T H String,
                    userName    :: T H String,
                    userEmail   :: T L String,
                    screenName  :: T L String}
```

With this implementation of *FormData*, Bob's old code will no longer be type correct. This is because *userName* now returns a *String* protected at H, but the return type of the *mkDBEntries* expects a $T\ L\ String$. As noted previously, Bob has no way to go from a $T\ H\ String$ to a $T\ L\ String$. The type system guarantees that the real name of the user is kept safe. He is forced to rewrite his implementation of *mkDBEntries* as follows.

Bob

```
mkDBEntries :: FormData →
  (T L [DBEntry], T H [DBEntry])
mkDBEntries form =
  (sequence [screenName form, userEmail form]
  , sequence [hash (password form), userName form])
```

The $sequence :: [T\ \ell\ a] \rightarrow T\ \ell\ [a]$ function collects the results produced by a list of protected computations and it is implemented as a derived operation, i.e., it does not break DCC abstractions. Observe that the name of the user is now in the correct list of database entries.

Monads in Haskell and super-bind. To convince Haskell's compiler that $T\ \ell$ is a monad, it is necessary to provide a *Monad* typeclass instance for $T\ \ell$, i.e., we need to implement the *return* and $(\ggg\!=)$ operators over that type. As claimed above, super-bind is general enough to implement $(\ggg\!=)$. Figure 6 shows that, by specializing the type of $(\ggg\!=)$, each family member $T\ \ell$, for a certain ℓ, can indeed be made an instance of the *Monad* type class.

Type errors. The reader may be concerned that the use of advanced features of the Haskell type system, like closed type families and data kinds give rise to complicated and difficult to interpret type errors. Indeed, this appears to be the case, in the worst cases type errors are on the form `"Could not unify 'False with 'True, arising from the use of >>>="`. However, recent work by Serrano and Hage [22] provides methods for improving type errors in Haskell programs by defining domain specific type errors. We note the incorporation of such techniques into our encoding of DCC as future work.

$$
\begin{aligned}
m \gg\!= f &\equiv \mathit{join}\ (\mathit{fmap}\ f\ m)\\
m \gg\!= \mathit{id} &\equiv \mathit{join}\ m\\
m \gg\!= (\mathit{return} \circ f) &\equiv \mathit{fmap}\ f\ m
\end{aligned}
$$

Figure 7: Laws relating $(\gg\!=)$ and *return* with *join* and *fmap*.

$$
\begin{aligned}
&\mathit{join}^{\text{DCC}} :: (\ell \preceq s) \sim \mathit{True} \Rightarrow T\ \ell\ s \to s\\
&\mathit{join}^{\text{DCC}}\ t = t \ggg\!= \mathit{id}\\
&\mathit{fmap}^{\text{DCC}} :: (\ell \sqsubseteq \ell') \sim \mathit{True} \Rightarrow (s \to t) \to T\ \ell\ s \to T\ \ell'\ t\\
&\mathit{fmap}^{\text{DCC}}\ t\ f = t \ggg\!= (\mathit{return} \circ f)
\end{aligned}
$$

Figure 8: Types for $\mathit{join}^{\text{DCC}}$ and $\mathit{fmap}^{\text{DCC}}$.

Discussion. The experienced functional programmer might propose the following combinator-based interface as an alternative to our formulation of DCC.

$$
\begin{aligned}
&\mathit{upgrade} :: (\ell \leqslant h) \sim \mathit{True} \Rightarrow T\ \ell\ a \to T\ h\ a\\
&\mathit{split} \quad :: T\ \ell\ (a, b) \to (T\ \ell\ a, T\ \ell\ b)
\end{aligned}
$$

The advantage of such an approach would be that it makes the security-related properties of the T monad family more explicit. However, it is not obvious what a sound and complete set of such combinators would look like. Furthermore, basing the translation on such combinators rather than our direct syntactic translation nullifies our argument that the translation is correct by construction. One observation is that any correct combinator-based translation should be implementable using our translation. For example, the *upgrade* and *split* combinators are trivially implementable using super-bind, thereby providing an interface which would be familiar to functional programmers while retaining the confidence provided by the syntactic translation.

Another alternative is the use of generalized monads [5] to keep track of the dependencies. Possibly employing a type like:

$$
(\gg\!=) :: T\ \ell\ a \to (a \to T\ \ell'\ b) \to T\ (\ell \sqcup \ell')\ b
$$

However, this would represent a significant departure from the original DCC formalism and we therefore consider an analysis of this approach outside the scope of this paper. The interested reader may consult [7] for a detailed discussion of this alternative approach.

4 AN ALTERNATIVE FORMULATION

It is known that every computations involving a monad m may be built by (at least) two different interfaces, one using

$$
\begin{aligned}
&\mathit{return} :: a \to m\ a\\
&\mathit{fmap} \ :: (a \to b) \to m\ a \to m\ b\\
&\mathit{join} \ :: m\ (m\ a) \to m\ a
\end{aligned}
$$

and and the other using

$$
\begin{aligned}
&\mathit{return} :: a \to m\ a\\
&(\gg\!=) \ :: m\ a \to (a \to m\ b) \to m\ b
\end{aligned}
$$

Every interface has its pros and cons and programmers are likely to use both—we refer the interested reader to [34] for further details. In this light, given the non-standard notion of $(\ggg\!=)$, a natural question is what such an isomorphism would look like for DCC? Before we start answering that question, Figure 7 recaps the laws establishing the correspondence between these two formulations.

We start by finding the equivalent notion of *join* and *fmap* in DCC. However, beare in mind that we expect to end up with a "super-*join*" and "super-*fmap*" so we will refer to them as $\mathit{join}^{\text{DCC}}$ and $\mathit{fmap}^{\text{DCC}}$, respectively. The code for these functions are given in Figure 8. These functions are simply computed by replacing all occurrences of $(\gg\!=)$ with $(\ggg\!=)$, *join* with $\mathit{join}^{\text{DCC}}$, and *fmap* with $\mathit{fmap}^{\text{DCC}}$ in Figure 7. The derived $\mathit{join}^{\text{DCC}}$ inherits the constraint from $(\ggg\!=)$ that checks if a certain type s is protected by ℓ—i.e., $(\ell \preceq s) \sim \mathit{True}$. This function basically removes the $(T\ \ell)$-monad when the value that it protects is already protected by ℓ. The derived $\mathit{fmap}^{\text{DCC}}$, on the other hand, allows us to map a function over the content of the $(T\ \ell)$-monad but potentially returning the result in a more sensitive family member $T\ \ell'$, where $\ell \sqsubseteq \ell'$.

Unfortunately, we need to go through some contortions when using $\mathit{join}^{\text{DCC}}$ and $\mathit{fmap}^{\text{DCC}}$ in a way similar to how we would use *join* and *fmap*, respectively. To illustrate this point, we consider the case where we have a computation $t :: T\ H\ (T\ L\ a)$ that we want to "join" (collapse) in order to return something of type $T\ H\ a$. We then proceed to use super-*join* as follow.

$$
\begin{aligned}
&t' :: T\ H\ a\\
&t' = \mathit{join}^{\text{DCC}}\ (\mathit{join}^{\text{DCC}}\ (\mathit{fmap}^{\text{DCC}}\ (\mathit{fmap}^{\text{DCC}}\ \mathit{return}_H)\ t))
\end{aligned}
$$

Observe that the problem with this example is that $T\ L\ a$ is not protected by H, therefore, we cannot simply apply $\mathit{join}^{\text{DCC}}$—the type constraint $H \preceq (T\ L\ a)$ does not hold! Instead, we need to transform t into something of type $T\ H\ (T\ L\ (T\ H\ a))$ by calling $\mathit{fmap}^{\text{DCC}}\ (\mathit{fmap}^{\text{DCC}}\ \mathit{return}_H)\ t$. After that, we take the two inner T's, i.e., $T\ L\ (T\ H\ a)$, and collapse them to $T\ H\ a$—this is done by the inner $\mathit{join}^{\text{DCC}}$. To conclude, the outer $\mathit{join}^{\text{DCC}}$ takes something of type $T\ H\ (T\ H\ a)$ and collapses it to $T\ H\ a$ as wanted.

As types become more complex, for example when we have a term t of type $T\ H\ (T\ L\ a, T\ H\ b)$, we would expect to be able to use $\mathit{join}^{\text{DCC}}$ to get a (collapsed) term t' of type $(T\ H\ a, T\ H\ b)$, but once again we arrive at a verbose implementation.

$$
\begin{aligned}
&t' :: (T\ H\ a, T\ H\ a)\\
&t' = \mathit{join}^{\text{DCC}}\ (\mathit{fmap}^{\text{DCC}}\\
&\qquad (\lambda(t_l, t_h) \to (\mathit{join}^{\text{DCC}}\ (\mathit{fmap}^{\text{DCC}}\ \mathit{return}_H\ t_l), t_h))\ t)
\end{aligned}
$$

As in the example above, we need to convert the first component of the pair from something of type $T\ L\ a$ to something of type $T\ L\ (T\ H\ a)$ to be able to collapse it to something of type $T\ H\ a$. Once that is done, the outer $\mathit{join}^{\text{DCC}}$ collapses $T\ H\ (T\ H\ a, T\ H\ a)$ to $(T\ H\ a, T\ H\ a)$ as expected.

All these contortions arise from having public values wrapped in a sensitive T-family member, which makes them effectively non-public values. To convince the type system that this is the case,

```
class Upgrade a b where
  up :: a → b
instance (ℓ ⊑ ℓ')~True ⇒ Upgrade (T ℓ a) (T ℓ' a) where
  up t = t >>= return
```

Figure 9: A type class for automatic upgrading.

```
join^DCC* :: (Upgrade s s', (ℓ ≤ s)~True) ⇒ T ℓ s → s'
join^DCC* t = t >>= up
```

Figure 10: $join^{DCC^*}$ with automatic upgrading.

we need to take those public values and *upgrade* them to sensitive ones, i.e., transforming them from something of type $T\ L\ a$ into something of type $T\ L\ (T\ H\ a)$ with $fmap^{DCC}$. After that, the $(T\ L)$-monad can be removed using $join^{DCC}$. To simplify the life of the programmer, we implement the type class *Upgrade a b* in Figure 9. The type class provides the (overloaded) operators $up :: a \to b$ which precisely implements the procedure for upgrading values of type a into values of type b to have potentially more sensitive labels. The most interesting cases for *up* is the instance involving the T-monad family present in Figure 9 —see Appendix A.1 for more details. In that case, to upgrade $T\ \ell\ a$ into $T\ \ell'\ a$, it must be the case that ℓ' is at least as sensitive as ℓ ($\ell \sqsubseteq \ell'$). Observe that the instance demands the type constraint $(\ell \sqsubseteq \ell')\sim True$, which arises from the use of $(\gg\!=)$.

With the type class *Upgrade* in place, we proceed to change the definition of $join^{DCC}$ to automatically adjust labels when needed—see Figure 10. Different from $join^{DCC}$, $join^{DCC^*}$ allows for a more elegant code. We revisit the examples shown before. When we have a term $t :: T\ L\ (T\ H\ a)$, we can transform it to a term of type $t :: T\ H\ a$ as follows.

```
t' :: T H a
t' = join^DCC* t
```

Similarly, when we have a term of type $T\ H\ (T\ L\ a, T\ H\ b)$, we can make an expression of type $(T\ H\ a, T\ H\ b)$ as follows.

```
t' :: (T H a, T H b)
t' = join^DCC* t
```

The equations in Figure 7 hold for $join^{DCC^*}$ provided that the type constraint *Upgrade s s'* can be satisfied. For instance, if the *up* method does not change the type, that is to say *Upgrade s s* holds, then we have exactly the same equations that we had previously.

It is also possible to derive $(\gg\!=)$ from $join^{DCC^*}$ and $fmap^{DCC}$. This encoding requires $join^{DCC^*}$ and $fmap^{DCC}$ to use unT^{TCB} in their definitions. A natural questions is then if such occurrences of unT^{TCB} are sound—i.e., it does not provide opportunities to leak information. We argue that, as long as the equations in Figure 7 hold for the DCC variants, it is possible to translate every occurrence of $join^{DCC^*}$ and $fmap^{DCC}$ back into $(\gg\!=)$, which in turn uses unT^{TCB} safely due to its type constraint. We leave the primitive definitions of $join^{DCC^*}$ and $fmap^{DCC}$ as well as the derivation of $(\gg\!=)$ as an exercise for the reader.

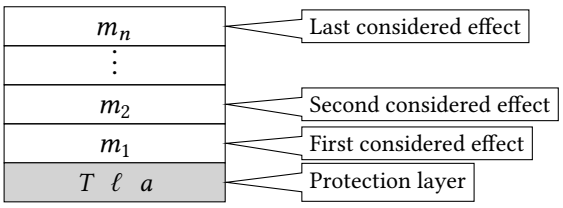

Figure 11: $(T\ \ell)$-monad at the base of monadic stacks

```
type Outputs o
type WT o ℓ a = WriterT (Outputs o) (T ℓ) a
```

Figure 12: Adding outputs.

5 SIDE-EFFECTS

In this section, we show how to extend DCC to handle effects. Being a family of identity monads, T does not provide any interesting side-effect on its own. At a first glance, it is not clear that effects can be naturally handled by DCC without having to trust the implementation of each and every effect. Approaches for information-flow control in Haskell have typically *intertwined security concerns with monadic effects*, i.e., side-effects are performed by primitive operations of their respective security monads (see for example [18, 19, 26]). Instead, we show that, as long as side-effects can be mimicked or denoted by pure computations [33], they can be added "on top" of our implementation of DCC. In other words, we show that the implementation of such effects need not be inherently trusted, but may be added to a monad of type $T\ \ell$ without using unT^{TCB}—the only primitive that must be handled with care. The main idea that makes this possible is to place the $(T\ \ell)$-monad at the bottom of a stack of monads (where each layer implements an effect). By doing so, data handled by upper layers is wrapped, i.e., protected, by $T\ \ell$. Figure 11 depicts this idea.

The rest of the section shows the derivation process of securely adding the most common effects to DCC, namely outputs, state, and error handling. For simplicity, we focus initially on effects provided by the *Writer* monad, a monad which equips computations with outputs.

5.1 Outputs

To start adding outputs, we resort to one of the most frequent tools that Haskell programmers rely on: *monad transformers* [16]. In a nutshell, a monad transformer provides effects similarly to regular

```
class Project (ℓ :: Lattice) where
  type SegmentsAbove ℓ o
  project :: Segments o → (T ℓ o, SegmentsAbove ℓ o)
  merge   :: (T ℓ o, SegmentsAbove ℓ o) → Segments o
             → Segments o
```

Figure 13: The *Project* type class.

monads, however a monad transformer cannot stand on its own; instead, it modifies the behavior of an underlying monad by means of pure functions to incorporate the effects associated with the transformer. Figure 12 shows the interface for *WT* (the *Writer-T* monad). The *WT o ℓ a* monad is built by applying the monad transformer *WriterT* to *T ℓ*, creating the type *WriterT* (*Outputs o*) (*T ℓ*) *a*, which extends the (*T ℓ*)-monad with outputs of type *Outputs o*. In IFC systems, end-points get associated with a security level denoting the sensitivity of the data entering or leaving the system. Having classified the sensitivity of outputs, it is possible to detect when information might get leaked, e.g., when a program tries to produce a sensitive output in the public partition or part of the system. Hence, we consider outputs as partitioned per security level—this is exactly what the type *Outputs o* denotes. For the two-point lattice, outputs are partitioned into two segments: one for sensitive and one for public outputs—observe that every partition is protected by *T*.

```
type Segments o = (T L o, T H o)
type Outputs o  = Segments o
```

We use the name *Segments* instead of *Outputs* as the idea of partitioning data by security level is more general than outputs and we will show how it extends when adding state as side-effect later.

In the writer monad, the operation producing outputs is the function *tell* :: *MonadWriter o m* ⇒ *o* → *m* (), where *m* is the type of the writer monad and *o* is the type of the output being produced. What would be a secure version of *tell* ? Intuitively, it should be the case that taking something at confidentiality level ℓ_L, it should be only possible to write in the output partition label with ℓ_H, where ℓ_H is the same as or higher than ℓ_L, that is $\ell_L \sqsubseteq \ell_H$. This is known as the no-write-down policy [3]. A computation is secure as long as it does not write data to less sensitive sinks. We will show that DCC guides us to a secure implementation of *tell*, called $tell^{WT}$, which implements precisely the no-write-down policy. To help readers, we indicate the relationship between type variables in their subindexes, we use type variables ℓ_L and ℓ_H to attest that $\ell_L \sqsubseteq \ell_H$. Before diving into the details of $tell^{WT}$, we need to implement some auxiliary functions.

For the rest of the section, we need to consider that outputs have a monoid structure, i.e., they have both a neutral element and an (associative) operations to concatenate outputs—see details in Appendix A.2. Figure 13 shows an auxiliary type class needed to provide secure outputs. The type class provide operations to *manipulate the space of outputs which are, at least, as sensitive as a given security level ℓ*. The associated type *SegmentsAbove ℓ* defines the

```
instance Project H where
  type SegmentsAbove H o = ()
  project (_, h)         = (h, ())
  merge   (h, ()) (ℓ, _) = (ℓ, h)
instance Project L where
  type SegmentsAbove L o = T H o
  project (ℓ, h)    = ( ℓ, h)
  merge   (ℓ, h) _  = ( ℓ, h)
```

Figure 14: Instances for *Project H* and *Project L*

```
tell^WT :: (Monoid o, Monoid (SegmentsAbove ℓ o)
           , Project ℓ o) ⇒ T ℓ o → WT o ℓ' ()
tell^WT t = tell (merge t mempty)
```

Figure 15: Secure version of primitive *tell*.

structure of all the levels strictly above *ℓ*. The *project* function takes the outputs and returns the partitions at level *ℓ* and higher. The *merge* function, on the other hand, takes all segments (*Segments o*) and replaces the values at level *ℓ* and higher by the first argument (*T ℓ o*, *SegmentsAbove ℓ o*). To illustrate how this type class works, Figure 14 shows instances useful for the two-point lattice, where outputs have type *o*. The instance *Project H* indicates that there are no outputs strictly more sensitive than *H* by defining type *SegmentsAbove H o* as unit, i.e., (). Similarly, the instance *Project L* indicates that the memory strictly above *L* is of type *T H o* by defining *SegmentsAbove L o* as *T H o*. The implementation for *project* and *merge* of both instances are self-explanatory. Note that, so far, we do not need to trust any other aspects of our code than what we trusted before—we have not used unT^{TCB} at all!

Figure 15 shows the secure, *derived*, implementation of *tell*. The type constraints in the type signature of $tell^{WT}$ reflect that outputs and partitions have a monoid structure—see constraints *Monoid o* and *Monoid* (*SegmentsAbove ℓ o*), respectively. The constraint *Project ℓ o* arising from the use of *merge* in the definition of $tell^{WT}$. This function is simply defined as merging the output *t* into an empty output (*mempty*). Ostensibly, the type signature looks insecure, i.e., the argument to $tell^{WT}$ is protected at *ℓ* (*T ℓ o*) but the resulting computation handles (protects) data at arbitrary label *ℓ'* (*WT o ℓ'* ())—recall the definition of *WT* in Figure 12. However, precisely because the argument is of type *T ℓ o*, it means that it will be *always* protected at *ℓ*, no matter in what context we attempt to output it. As a result, calls to $tell^{WT}$ with public arguments in sensitive contexts are allowed but no effects would be ever performed. To illustrate this point, we proceed to describe some examples.

Example 5.1. Consider the code below.

$$
\begin{aligned}
&\mathit{cast}^{\mathrm{WT}} :: (\mathit{Monoid}\ o, \mathit{Project}\ \ell_{\mathrm{H}}\ o,\\
&\qquad (\ell_{\mathrm{H}} \preceq \mathit{OutputsAbove}\ \ell_{\mathrm{H}}\ o) \sim \mathit{True})\\
&\Rightarrow \quad \mathit{WT}\ o\ \ell_{\mathrm{H}}\ a \to \mathit{WT}\ o\ \ell_{\mathrm{L}}\ (T\ \ell_{\mathrm{H}}\ a)
\end{aligned}
$$

Figure 16: Primitive $\mathit{cast}^{\mathrm{WT}}$

```
secure :: Monoid o ⇒ T L o → T H Bool → WT o H ()
secure output secret = do bool ← lift secret
                          when bool (tell^WT output)
```

Where the $\mathit{lift} :: T\ \ell\ a \to \mathit{WT}\ o\ \ell\ a$ function unlabels a $T\ \ell$ value at the correct security level in the *WT* monad. The example seems to be insecure: depending on the value of the secret (*bool*), the program produces a public output ($\mathit{tell}^{\mathrm{WT}}$ *output*). However, since the resulting type is $\mathit{WT}\ o\ H$ (), it means that all the outputs are indeed protected by H—thus, the code does not leak information! In other words, once in the monad $\mathit{WT}\ o\ H\ a$, any public output has no effect since the whole output domain is protected by H.

The reason why the example above is secure relies one the fact that monad transformers are implemented using pure functions and that the $T\ \ell$-monad is placed at the bottom of the monadic stack. By definition, the monad transformer *WriterT* (as most monad transformers do) manipulates the underlying monad to introduce the output effect. More concretelly, the type $\mathit{WT}\ o\ H\ a$, i.e., $\mathit{WriterT}\ (\mathit{Outputs}\ o)\ (T\ H)\ a$, is isomorphic to the type $T\ H\ (a, \mathit{Outputs}\ o)$. Observe how the monad transformer leverages on the $T\ H$-monad to carry around outputs. This fact implies that all the outputs are considered sensitive by $\mathit{WT}\ o\ H\ a$. Then, it is not surprising that $\mathit{tell}^{\mathrm{WT}}$ can be called with a public argument in sensitive contexts; after all, outputs are effectively secret!

Example 5.2. Consider instead the following code.

```
secure :: Monoid o ⇒ T H o → T L Bool → WT o L ()
secure output public = do bool ← lift public
                          when bool (tell^WT output)
```

This function is very similar to that in the example above, except that the value to output is now sensitive ($T\ H\ o$), while the boolean ($T\ L\ \mathit{Bool}$) and the writer monad are not ($\mathit{WT}\ o\ L$ ()). As before, the code is safe to execute, i.e., a sensitive output gets performed based on the value of a public boolean.

One could still argue that the type of $\mathit{tell}^{\mathrm{WT}}$ should restrict the programmer and give a type error in the case where $\ell \not\preceq \ell'$ in order to hint about possible mistakes in the code. However, this would have no impact on security. The generality of $\mathit{tell}^{\mathrm{WT}}$ is comfortable; it allows for *securely* and indiscriminately writing to different outputs at different levels.

5.2 Label creep

Although the interface provided by $\mathit{tell}^{\mathrm{WT}}$ is secure, it leaves some things to be desired. As illustrated by Example 5.1, computations depending on sensitive data force the entire computation to be protected at H. In other words, *after inspecting a secret, public outputs are tainted with the label of the secret.* This phenomenon is known as *label creep*: the label associated with the computation reaches a point where it is not possible to perform any useful side-effects—even if the effects are benign and do not attempt to leak information. We illustrate the label creep problem in the following example.

Example 5.3. Even though the following example is well typed, and therefore secure, the intended public output on the final line ($\mathit{tell}^{\mathrm{WT}}$ *somethingPublic*) occurs inside the ($\mathit{WT}\ o\ H$)-monad. Consequently, it is considered secret.

```
creep :: Monoid o ⇒ T H Bool → WT o H ()
creep th = do
  secret ← lift th
    -- From now on, no more public outputs
  when secret (tell^WT somethingSecret)   -- secret output
  tell^WT somethingPublic   -- intended public output
```

To mitigate the label creep problem, we introduce another *derived* function called $\mathit{cast}^{\mathrm{WT}}$. Figure 16 shows the type signature for $\mathit{cast}^{\mathrm{WT}}$. Starting with the types, the *Monoid o* constraint arises from the use of the *Writer* effect, while the other two deserve some explanation. The constraint $\mathit{Project}\ \ell_{\mathrm{H}}\ o$ tells us that some manipulation on the sensitive part of the outputs takes place in $\mathit{cast}^{\mathrm{WT}}$ (explained below). Finally, the constraint $(\ell_{\mathrm{H}} \preceq \mathit{OutputsAbove}\ \ell_{\mathrm{H}}\ o) \sim \mathit{True}$ is redundant but appears in the type signature due to the inability of Haskell to figure out that ℓ_{H} protects all the outputs about itself.

The function $\mathit{cast}^{\mathrm{WT}}$ runs a computation in the ($\mathit{WT}\ o\ \ell_{\mathrm{H}}$)-monad (given as argument) and injects the produced sensitive outputs as part of the outputs handled by the resulting ($\mathit{WT}\ o\ \ell_{\mathrm{L}}$)-monad. It also protects the result of type a with label ℓ_{H}, that is $T\ \ell_{\mathrm{H}}\ a$. By applying $\mathit{cast}^{\mathrm{WT}}$, we do not need to remain in the ($\mathit{WT}\ o\ H$)-monad every time we need to inspect a secret. The role of $\mathit{cast}^{\mathrm{WT}}$ is to convert (cast) sensitive computations (at level ℓ_{H}) to public ones (at level ℓ_{L}) without leaking any information, while still retaining both the sensitive result and side-effects. Note that any public side-effects that may have been performed in the sensitive context are discarded by $\mathit{cast}^{\mathrm{WT}}$, this is possible because we are in a pure setting. In the jargon of more traditional IFC systems, $\mathit{cast}^{\mathrm{WT}}$ is an operation which allows to temporarily raised the *program counter* [32].

Figure 17 shows a graphic rendering of the core operational semantics of $\mathit{cast}^{\mathrm{WT}}$—we refer readers to Appendix A.3 for the concrete implementation. As a first step, $\mathit{cast}^{\mathrm{WT}}$ runs its argument of type $\mathit{WT}\ \ell_{\mathrm{H}}\ a$, which results in a result $x :: a$ and outputs of type $\mathit{Outputs}\ o$—see left part of the graphic. From the produced outputs, we identify three segments of outputs: the outputs at level

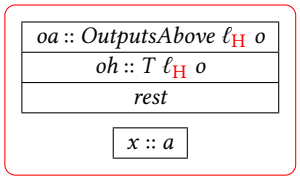

Figure 17: Semantics for $\mathit{cast}^{\mathrm{WT}}$

ℓ_H and higher, respectively denoted by variable $oh :: T\ \ell_H\ o$ and $oa :: OutputsAbove\ \ell_H\ o$, and outputs which are strictly less sensitive or incomparable to ℓ_H, denoted by *rest* in the graphic. Function $cast^{WT}$ then proceeds to deconstruct, by using super-bind, the returned value and the outputs ($\lambda(x, outputs) \rightarrow \ldots$). The outputs at level ℓ_H and above are extracted (*project outputs*), an action that produces the tuple $(oh, oa) :: (T\ \ell_H\ o, OutputsAbove\ \ell_H\ o)$. Hence, it disregards any other outputs (i.e., *rest*) and considers *only* those that can be safely affected by the computation of type $WT\ \ell_H\ a$, i.e., those outputs which do not write down from the security level ℓ_H. The key realisation behind the implementation of $cast^{WT}$ is that, while sensitive computations may depend on low data, they may not influence low outputs; a property ensure by the type system!

The idea of $cast^{WT}$ is common in other Haskell IFC libraries [18, 19, 26]. However, different from them, $cast^{WT}$ is a *derived* operation rather than a trusted primitive. We should remark that the core idea of $cast^{WT}$ is present in the appendix of the DCC paper [1] when showing an embedding of the imperative (stateful) SV-calculus in DCC. In the next section, we show that the core idea behind $cast^{WT}$ can be extended to cover other effects mimicked by pure expressions, namely, state and error handling.

5.2.1 Revisited running example. With $cast^{WT}$ in place, Alice and Bob can rewrite their application to make use of the power of monadic effects.

Example 5.4. As Alice decides to add more and more functionality to her website, she realises that a *Writer* effect is necessary to handle the many functions for generating database entries. Furthermore, Alice has decided to add a function $isSecure :: String \rightarrow Bool$ which checks if a password is deemed secure.

Bob

```
mkDBEntries :: FormData → WT [DBEntry] L (T H Bool)
mkDBEntries form = do
    -- public output
  tell^WT (sequence [screenName form, userEmail form])
    -- including a sensitive output
  cast^WT (mkSecretEntries form)
```

Bob

```
mkSecretEntries :: FormData → WT [DBEntry] H Bool
mkSecretEntries form = do
  pwd ← lift (password form)
  when (isSecure pwd)
    (tell^WT (sequence
      [fmap hash (password form), userName form]))
  return (isSecure pwd)
```

With this implementation, Alice can decide when she, in the trusted code, obtains the result of *mkDBEntries* and if she is to commit the result to the database based on the result in the $T\ H\ Bool$ coming from the action *return sec*. Note that $cast^{WT}$ allows Bob to write the sensitive part of the code as a stand-alone module

```
  -- Definition
type WET o ℓ a =
  WriterT (Outputs o) (EitherT String (T ℓ)) a
  -- Outputs
tell^WET :: (Monoid o, Project ℓ o) ⇒ T ℓ o → WET o ℓ' ()
  -- API for error handling
throwError :: Monoid o ⇒ String → WET o ℓ ()
catchError :: Monoid o ⇒
  WET o ℓ a → (String → WET o ℓ a) → WET o ℓ a
  -- Label creep
cast^WET :: (Monoid o, Project ℓ_H o,
        (ℓ_H ≤ OutputsAbove ℓ_H o)~True)
  ⇒ WET o ℓ_H a → WET o ℓ_L (T ℓ_H (Either Error a))
```

Figure 18: Combination of exception and writer effects.

6 COMBINING EFFECTS

In this section, we show that it is possible to derive secure operations capable of handling multiple effects via monad transformers.

6.1 Error handling

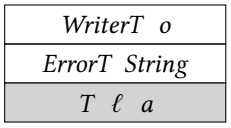

Figure 19: The monadic stack

We begin by extending the *WT*-monad with exceptions to create the *WET*-monad (where the *E* stands for errors)—see Figure 18. The type $WET\ o\ \ell\ a$ is a monad stack which consists on the $(T\ \ell)$-monad at the bottom, the monad transformer $ErrorT\ String$ in the middle (which extend the $(T\ \ell)$-monad with error messages of type *String*), and $WriterT\ (Outputs\ o)$ at the top—Figure 19 depicts the monadic stack.

The interface for *WET* is similar to that of *WT*, i.e., it provides the primitive $tell^{WT}$. It also provides the functionality for throwing (*throwError*) and catching (*catchError*) errors. The key difference between the interface of *WT* and *WET* lies in the implementations of $cast^{WT}$ and $cast^{WET}$. The derived operation $cast^{WET}$ requires checking if the sensitive computation threw an error—an error that must be kept secret ($T\ \ell_H\ (Either\ Error\ a)$). This occurs because the monad family *T* keeps us from communicating errors from a sensitive (ℓ_H) to a less sensitive (ℓ_L) context. Since the error might occur within the $(WET\ o\ \ell_H)$-monad, the error must then remain protected by ℓ_H. This restriction is in agreement with how other IFC libraries deal with exceptions [18, 26]. Differently from such libraries, we derive how to securely deal with errors by simply using $T\ \ell$ and the DCC typing rules.

```
  -- Definition
type ST s ℓ a = StateT (State s) (T ℓ) a
  -- State
get :: ST s ℓ (State s)
put :: State s → ST s ℓ a
  -- Avoding label creep
cast^ST :: (Project ℓ_H s,
            (ℓ_H ≤ StateAbove ℓ_H s)~True)
         ⇒ ST s ℓ_H a → ST s ℓ_L (T ℓ_H a)
```

Figure 20: Secure stateful computations.

```
  -- Definition
type SET s ℓ a = StateT s (EitherT String (T ℓ)) a
  -- State
get :: SET s ℓ (State s)
put :: State s → SET s ℓ a
  -- Label creep
cast^SET :: (Project ℓ_H,
             (ℓ_H ≤ (StateAbove ℓ_H s))~True))
          ⇒ SET s ℓ_H a → SET s ℓ_L (T ℓ_H (Either String a))
```

Figure 21: Combining state and error handling.

6.2 State

Another effect which can be implemented on top of DCC is state. In order to indicate that the focus is now on state, we change the type variable o denoting "outputs" for type variable s denoting "state". We also introduce type synonyms for *Segments* s and *SegmentsAbove* ℓ s as *State* s and *StateAbove* ℓ s, respectively.

Figure 20 shows the definition and interface of stateful computations as the $(ST\ s\ \ell)$-monad. It is simply the application of the stateful (*StateT*) monad transformer over *State* s and $T\ \ell$. The interface presents specific primitives to fetch (*get*) and store state (*put*). The type signature for $cast^{ST}$ is exactly the same as $cast^{WT}$ except for mentioning *ST* and *StateAbove* instead of *WT* and *OutputsAbove*, respectively—Appendix A.4 shows the complete, *derived*, implementation of $cast^{ST}$.

As the *WT* monad can be extended to handle errors, so it can be the *ST* monad. Figure 21 shows the interface for the $(SET\ s\ \ell)$-monad, a monad with state and error handling. The idea behind $cast^{SET}$ is the same as $cast^{WET}$, catch the error and put it in the result, adjusting the effect accordingly (depending on *EitherT* being on the outside or inside of *StateT*).

6.3 I/O

We explore if DCC is capable of securing effects provided by one of the most popular monads in Haskell, i.e., the *IO* monad. This monad provides the API to perform effects via the underlying I/O system, e.g., the file system, network communication, running executables, etc. The *IO* monad, different from the ones we have seen so far, is opaque and lacks a manner to "get out of it" without possibly breaking type-safety [29]—i.e., there is no safe function of type $IO\ a \to a$. Once a computation has type *IO*, it will remain there.

We argue that DCC can only secure effects which can be represented in a pure way (like state, exceptions, etc.), but not the ones provides by the (opaque) *IO* monad. To illustrate this point, let us consider *IO*-effects conceived inside the $(T\ H)$-monad, i.e., a computation of type $T\ H\ (IO\ a)$. Are such *IO*-effects safe to perform? The answer is no, we have *no guarantee* how such effects were constructed or what they do! If those effects were internal to the program, like state or exceptions, they could potentially be managed as described before, i.e., building a monadic stack where the $T\ \ell$-monad is at the bottom. However, *IO*-effects could also involve external communication or any other type of effects. For instance, they could observe sensitive information and send it over the network.

One could be tempted to conceive a primitive of the form

```
secureIO :: T H (IO a) → IO (T H a)
```

where the *IO*-actions are being taking out of the $T\ H$-monad—after all, they cannot be observed (the *IO* monad is opaque). Unfortunately, *secureIO* does not honor to its name. Consider the following leaking program.

```
attack :: T H Bool → T H (IO Bool)
attack tb =
tb >>= λb → do if b then putStrLn "True"
                    else putStrLn "False"
              return_H (return b)
leak :: T H Bool → IO (T H Bool)
leak = secureIO ∘ attack
```

This code always returns the same boolean wrapped in the *IO*-monad (*return* b), while it performs two different effects depending on the value of the secret boolean b—i.e., either showing `"True"` or `"False"` on the screen. Consequently, when applying *secureIO* and taking the *IO*-effects "out" of $T\ H$, the attacker can execute them and infer the value of b from reading the console—which is a public channel. This argument has been previously brandished to justify the introduction of another security monad capable to handle *IO*-effects [18, 19, 26]. In this work, however, we restrict ourselves to only having the *T*-monad family.

Alternatively, it could be argue that there could be a pure representation of the *IO*-effects rather than the *IO*-monad itself [27]. Having that, it is just a matter of placing $(T\ \ell)$-monads in the pure representation of actions according to the sensitivity of inputs and outputs. We declare this intriguing idea as future work.

7 RELATED WORK

DCC. The seminal work on DCC [1] has inspired a large amount of work. Tse and Zdancewic translated DCC into System F [30].

They conceived the rules for () and use parametricity to prove non-interference. Authors also provided an implementation of DCC in Haskell [30]. However, this implementation differs from ours in two main aspects. First, the $T\ \ell\ a$ monad is isomorphic to the type $\ell \to a$ rather than an identity monad as originally conceived in DCC [1]—this representation is inspired by the translation of DCC into System F. Secondly, the super-bind operator is implemented as an overloaded primitive (type-class). As a result, it is necessary to give an implementation of super-bind *for each possible type constructor* present in DCC. Instead, we provide a single implementation of super-bind and provide a type-level function to enforce the corresponding security checks—which aligns with having a single typing rule for super-bind in DCC. Shikuma and Igarashi gave a counterexample to show that Tse and Zdancewic translation was not sound [23, 24]. Instead, Shikuma and Igarashi propose a calculus very much inspired by DCC, called *sealing calculus*. To make their translation proof work, they weaken the source language, i.e., admit some additional well-typed terms compared with DCC, at the same time strengthening the target language to be a restricted version of System F. Recently, Bowman and Ahmed show a sound translation from DCC into System F by the use of existential types [4]—thus being the first ones to prove non-interference by encoding it in parametricity. As in [4, 23, 24], our work focuses on the terminating fragment of DCC. We carefully craft an implementation which looks as a syntactic translation of DCC rules. By doing so, we can rely on the results of [4] to give us confidence about the correctness of our code, where parametricity lies behind polymorphic languages like Haskell.

Security libraries. Many monadic IFC security libraries, e.g., LIO [26], HLIO [6], and MAC [18], provide security monads which *intertwine pure and effectful computations*—this is one of the main differences with our work. Instead, we describe a methodology for using DCC as a foundational approach to secure effects mimicked by pure computations via the use of monad transformers. As a result, and contrary to existing IFC libraries, we are able to *derive* (by complying with the type system) a secure API to safely support exceptions, references, and any other effects mimicked by pure computations. On the other hand, IFC libraries (e.g., LIO) often come accompanied with proof showing that programs satisfy the non-interference security policy. We argue that providing soundness proofs about our encoding would end up in essentially the same semantics/proofs as in the original DCC paper due to the nature of our implementation, i.e., a syntactic translation. (We note that stacks of monad transformers can be modeled in DCC by leveraging sum- and product-types.) This work also identifies DCC's limits: handling arbitrary I/O effects (see Section 6.3). In that situation, it is required to introduce security-specific monads much along the lines of what SecLib, LIO, and MAC do. Rajani et al. [17] study the trade-offs and equivalences between statically tracking dependencies using monads with more fine-grained traditional IFC type systems. Devriese and Piessens provide a monad transformer to extend imperative-like APIs with support for IFC in Haskell [9]. Similar to our work, they use type families to encode the security lattice; however, they provide *open* definitions for type-level functions. In principle, it would be possible to rephrase some of their static analyzes into our encoding of DCC. Jaskelioff and Russo implements a library which dynamically enforces IFC using secure multi-execution (SME) [13]—a technique that runs programs multiple times (once per security level) and varies the semantics of inputs and outputs to protect confidentiality. Rather than running multiple copies of a program, Schmitz et al. provide a library with *faceted values* [21], where values present different behavior according to the privilege of the observer. These last two libraries apply different ideas than those behind DCC: it enforces IFC dynamically by repeating computations rather than tracking dependencies.

8 FINAL REMARKS

By leveraging closed type families and type classes, we describe a novel implemention of DCC in Haskell which looks like a direct translation of DCC's typing rules—an elegant outcome of this work. We clarify some aspects of DCC's non-standard bind operation from the perspective of a Haskell programmer. Furthermore, we also provide a type directed approach (the *cast* functions) to avoid the label creep problem present in DCC. Once embedded in Haskell, DCC can be used as a solid base to secure programs with effects like outputs, state, exceptions, or combinations thereof. For that, we utilize monad transformers, well known tools for combining effects. Most importantly, to secure such effects, we *only* need to trust our type system (via the use of SafeHaskell [28]) and the implementation of super-bind. It becomes now possible to start envisioning the construction of larger case studies and applications in Haskell, where DCC is the underlying security mechanism—an interesting direction for future work.

In a recent paper, Austin *et al.* [2] study how to secure imperative languages constructs by translating them into a pure functional language. While the paper focuses on dynamic techniques, the articles states *"... one can translate various static information-flow systems for imperative languages to the Dependency Core Calculus (DCC) [1] ..."* However, the only translation in [1] is an IFC system for a imperative language with state. This work completes that statement by showing how to handle a richer set of effects—by simply embedding DCC into Haskell and letting the type system guiding us.

Acknowledgments. We would like to thank Deepak Garg, Vineet Rajani, Marco Vassena, and the anonymous reviewers for their comments of this work. This work was funded in part by the Swedish research agency VR and in part by the GRACeFUL (grant agreement No 640954) project.

REFERENCES

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. 1999. A Core Calculus of Dependency. In *Proc. ACM Symp. on Principles of Programming Languages*. 147–160.
[2] Thomas H. Austin, Cormac Flanagan, and MartÃın Abadi. 2012. A Functional View of Imperative Information Flow. *Lecture Notes in Computer Science* (2012), 34âĂŞ49.
[3] David E. Bell and L. La Padula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation.* Technical Report MTR-2997, Rev. 1. MITRE Corporation, Bedford, MA.
[4] William J. Bowman and Amal Ahmed. 2015. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015.*
[5] Jan Bracker and Henrik Nilsson. 2016. Supermonads: one notion to bind them all. In *Proceedings of the 9th International Symposium on Haskell*. ACM, 158–169.

[6] P. Buiras, D. Vytiniotis, and A. Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM.

[7] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 289–301.

[8] D. E. Denning and P. J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513.

[9] D. Devriese and F. Piessens. 2011. Information flow enforcement in monadic libraries. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '11)*. ACM.

[10] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM.

[11] J.A. Goguen and J. Meseguer. 1982. Security policies and security models. In *Proc of IEEE Symposium on Security and Privacy*. IEEE Computer Society.

[12] D. Hedin and A. Sabelfeld. 2011. A Perspective on Information-Flow Control. In *Proc. of the 2011 Marktoberdorf Summer School*. IOS Press.

[13] M. Jaskelioff and A. Russo. 2011. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (LNCS)*. Springer-Verlag.

[14] B. W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (Oct. 1973).

[15] P. Li and S. Zdancewic. 2006. Encoding Information Flow in Haskell. In *Proc. of the IEEE Workshop on Computer Security Foundations*. IEEE Computer Society.

[16] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *In Proc. of the 22nd ACM Symposium on Principles of Programming Languages. ACMPress.*

[17] Vineet Rajani, Iulia Bastys, Willard Rafnsson, and Deepak Garg. 2017. Type systems for information flow control: the question of granularity. *SIGLOG News* 4, 1 (2017).

[18] A. Russo. 2015. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM.

[19] A. Russo, K. Claessen, and J. Hughes. 2008. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM.

[20] A. Sabelfeld and A. C. Myers. 2003. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19.

[21] Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. 2016. Faceted Dynamic Information Flow via Control and Data Monads. In *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings.*

[22] Alejandro Serrano and Jurriaan Hage. 2016. *Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules.* Springer Berlin Heidelberg, Berlin, Heidelberg, 672–698. https://doi.org/10.1007/978-3-662-49498-1_26

[23] Naokata Shikuma and Atsushi Igarashi. 2006. Proving Noninterference by a Fully Complete Translation to the Simply Typed lambda-Calculus. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers.*

[24] Naokata Shikuma and Atsushi Igarashi. 2008. Proving Noninterference by a Fully Complete Translation to the Simply Typed Lambda-Calculus. *Logical Methods in Computer Science* 4, 3 (2008).

[25] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. 2011. Disjunction Category Labels. In *Proc. of the Nordic Conference on Information Security Technology for Applications (NORDSEC '11)*. Springer-Verlag.

[26] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*.

[27] Wouter Swierstra and Thorsten Altenkirch. 2007. Beauty in the Beast: A Functional Semantics of the Awkward Squad. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*. 25–36.

[28] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe haskell. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 137–148.

[29] D. Terei, S. Marlow, S. Peyton Jones, and D. Mazières. 2012. Safe Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*. ACM.

[30] S. Tse and S. Zdancewic. 2004. Translating dependency into parametricity. In *Proc. of the Ninth ACM SIGPLAN International Conference on Functional Programming*. ACM.

[31] Stephen Tse and Steve Zdancewic. 2004. Translating Dependency into Parametricity. *SIGPLAN Not.* 39, 9 (Sept. 2004), 115–125.

[32] D. Volpano, G. Smith, and C. Irvine. 1996. A Sound Type System for Secure Flow Analysis. *J. Computer Security* 4, 3 (1996), 167–187.

[33] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, 61–78.

[34] Philip Wadler. 1995. Monads for functional programming. In *International School on Advanced Functional Programming*. Springer, 24–52.

[35] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proc. of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM.

A APPENDIX

A.1 Automatic relabeling

In this Appendix, we provide the rest of the instances for the type class *Upgrade*: pairs and functions. As the following code shows, these instances simply work by inductively traversing the type being upgraded.

```
instance (Upgrade s s', Upgrade t t')
         ⇒ Upgrade (s, t) (s', t') where
  up (s, t) = (up s, up t)

instance Upgrade s s'
         ⇒ Upgrade (t → s) (t → s') where
  up f = up ∘ f
```

Upgrading a pair ($up\ (s,t)$) is the same thing as upgrading both elements (i.e., $up\ s$ and $up\ t$)). Upgrading a function ($up\ f$) is the same as upgrading the result of the function ($up \circ f$).

A.2 Structure for outputs

As explain in the main text of this work, we consider outputs of type o as having a monoid structure. Consequently, we consider every output partition $T\ \ell\ o$ as a monoid too since we need both neutral elements, denoted by *mempty*, and a way to combine *protected* outputs from different computation—denoted by the primitive *mappend*.

```
instance Monoid o
         ⇒ Monoid (T ℓ o) where
  mempty          = return mempty
  mappend m₀ m₁ = do
    o₀ ← m₀
    o₁ ← m₁
    return (mappend o₀ o₁)
```

The neutral element is simply the *protected* computations which returns the neutral element of type o. Combining, i.e., "*mappend*-ing", two *protected* computations which produce os is done by running the first computation m_0 and then the second m_1 in order to return their *protected* combined results ($return\ (mappend\ o_0\ o_1)$).

A.3 Label creep

In this section, we provide the implementation of $cast^{\mathrm{WT}}$. As described before, the key observation for this primitive is that it requires getting rid of the low output created by running the high computation being casted while retaining the high part of the output. Specifically, the code for $cast^{\mathrm{WT}}$ is as follows.

```
cast^WT m = do
  let (highOutputs, tl_x) =
    runWriterT m >>= λ(x, outputs) → (project outputs, return x)
  tell (merge highOutputs mempty)
  return tl_x
```

First, it runs the sensitive computation m ($runWriterT\ m$). By using super-bind, it extracts the result (x) and produced outputs ($outputs$) by m—observe the function $\lambda(x, outputs) \rightarrow \ldots$). Then, $cast^{\text{WT}}$ only keeps the sensitive part of the outputs ($project\ outputs$) and the result produced by m ($return\ x$). Consequently, the pair ($highOutputs, tl_x$) respectivelly holds the sensitive outputs and the sensitive result produced by m. The next step by $cast^{\text{WT}}$ fills the tuple of empty segments of outputs ($mempty$) with the sensitive part produced by m by running $tell\ (merge\ highOutputs\ mempty)$. At this point, the produced ($WT\ o\ \ell_{\text{L}}$)-monad computation contains as sensitve output $highOutputs$ while its public segment is empty—nevertheless, other ($WT\ o\ \ell_{\text{L}}$)-monadic computations could populate such segment as shown by Example 5.4.

A.4 State

When considering state, the implementation of $cast^{\text{ST}}$ requires keeping track of what the secret computation has done to the state. While it is different from $cast^{\text{WT}}$, the typing rules of the T monad family keep us from leaking secrets! Below is the implementation of $cast^{\text{ST}}$:

$cast^{\text{ST}}\ m$ = **do**
  $s \leftarrow get$
  let (s_{H}, tl_x) =
    $runStT\ m\ s \ggg\!= \lambda(x, st) \rightarrow (project\ st, return\ x)$
  $put\ (merge\ s_{\text{H}}\ s)$
  $return\ tl_x$

The implementation of works by the following steps:

- Obtain the current state ($s \leftarrow get$).
- Run the sensitive computation with the current state as starting state ($runStT\ m\ s$) and obtain the segments of the resulting state which are protected at ℓ_{H}.
- Return the result (x) of the computation wrapped in $T\ \ell_{\text{H}}$.
- Merge the new, sensitive, state with the old public state—observe the instruction $put\ (merge\ s_{\text{H}}\ s)$.
- Return the result protected by $T\ \ell_{\text{H}}$ ($return\ tl_x$).

Similar as $cast^{\text{WT}}$, note the use of super-bind to obtain the value produced from the sensitive computation m into two parts—observe function ($\lambda(x, st) \rightarrow \ldots$). In this case, super-bind respectivelly produces the result of the sensitive computation ($return\ x$) and the sensitive part of the state ($project\ st$) which will be subsequently merged back into the state ($put\ (merge\ s_{\text{H}}\ s)$).